AD-A042 479 STANFORD RESEARCH INST MENLO PARK CALIF COMPUTER SCI--ETC F/G 9/2
THE RELATIONSHIP OF SYSTEM FAMILIES TO HDM.(U)
JUN 77 L ROBINSON N00123-76-C-0195

UNCLASSIFIED CSL-50 NL

| OF |
AD
A042479

END
DATE FILMED
8-77
DDC

(12)

Technical Report CSL-50
Deliverable A-013

THE RELATIONSHIP OF SYSTEM FAMILIES TO HDM

By: LAWRENCE ROBINSON

Prepared for:

NAVAL OCEAN SYSTEMS CENTER
SAN DIEGO, CALIFORNIA 92152

CONTRACT N00123-76-C-0195
SRI International Project 4828

W. LINWOOD SUTTON, Contract Monitor

STANFORD RESEARCH INSTITUTE
Menlo Park, California 94025 · U.S.A.

STANFORD RESEARCH INSTITUTE
Menlo Park, California 94025 · U.S.A.

(9) Technical Report CSL-50
Deliverable A-013

(11) June 1977

(12) 42p.

(14) CSL-50

(6) THE RELATIONSHIP OF SYSTEM FAMILIES TO HDM.

(10) By: LAWRENCE ROBINSON

Prepared for:

NAVAL OCEAN SYSTEMS CENTER
SAN DIEGO, CALIFORNIA 92152

(15) CONTRACT N00123-76-C-0195
SRI International Project 4828

W. LINWOOD SUTTON, Contract Monitor

Approved by:

JACK GOLDBERG, *Director*
Computer Science Laboratory

EARLE D. JONES, *Executive Director*
Information Science and Engineering Division

ABSTRACT

A system family is a set of computer systems that share common properties. This report discusses the use of an existing software development methodology, HDM (the SRI Hierarchical Development Methodology), to describe system families. It is believed that, if a group of related systems is developed as a family using this methodology, costs will be saved over separate development efforts. Several issues in system families are presented, and some ongoing work in applying this methodology is discussed.

CONTENTS

ILLUSTRATIONS

TABLES

ACKNOWLEDGMENTS

The author is indebted to Jack Goldberg, Karl Levitt, Ross Scroggs, and Lin Sutton for their comments and assistance in the preparation of this report.

I INTRODUCTION

A major goal of this project has been to develop a methodology for describing families of computer systems, in order to assist the Navy in the task of designing, implementing, and maintaining a multiplicity of such systems. Certain sets of computer systems have similarities in requirements, applications, or structure; such sets can be considered system families. If the similarities in the members of a system family could be taken advantage of, then it might be possible to reduce the total cost of developing the systems in the family. Costs could be reduced in the following ways:

* A given aspect of design, implementation, or maintenance could be addressed exactly once for all members of a system family, thus reducing the total development time.
* The same personnel could participate in the actual design, implementation, and maintenance of many systems in the family, without the need for retraining.
* A better understanding of the requirements of the system family could be achieved early in the development effort, thus reducing the number of unfortunate design decisions. Such errors in design can be extremely costly.
* Users might be able to operate many systems in the family without the need for retraining.
* Given an existing system family, new family members could be produced at low cost to meet particular operating environments.
* Systems could be transported economically to new hardware facilities as they become available.

The presence of a methodology for creating and describing system families would enable the development process to be geared to the production of families with a maximum number of similarities, thus reducing costs even further. Because of the extremely high cost of software production today, the investigation of system families can yield results that are potentially of great benefit to the Navy.

The approach originally proposed was to base the family methodology on a mathematically-based methodology for the design, implementation, and verification of large systems, already under development at SRI International. This methodology, called HDM (the SRI Hierarchical Development Methodology), is based on the ideas of formal specification and modular decomposition of Parnas ([1], [2], [3]), the ideas of structured programming and hierarchy of Dijkstra ([4], [5]), and the ideas on language design and program verification of Hoare ([6], [7]). Several documents on HDM exist, both outside and inside the current project work ([8], [9], [10], [11], [12]). Other researchers are also investigating the problem of system families, particularly Parnas [13] and Habermann [14], whose work is quite similar to the work underway at SRI International. The differences between the approach used here and related work will not be dwelt on in this document.

At the time of the proposal for this project, HDM was intended to aid in the development of a single system at a time. A valid question was whether it could be used for the development of entire families of systems, and if so how. At the time of the proposal, it was believed that HDM had many properties that were desirable for the description of system families, as well as single systems. Examples of such properties are the independence of implementation from specification and the ability to generalize a module specification to apply to many similar modules. It was also acknowledged in the proposal that some additions or changes to the specification language or concepts of HDM might be needed in order to apply it to system families. However, the additions and changes were minimal, consisting chiefly of ideas for new tools and a slightly different point of view.

This document discusses the following items:

* System families as described in terms of system properties
* The choice of a medium for describing system families
* The criteria for choosing system families
* The approach to describing system families in HDM

* Testing the approach to system families on the set of Navy message processing systems
* Some conclusions to be drawn from the work to date.

Sections II, III, and IV give a conceptual framework for the application of HDM to the family objective. Readers who prefer concrete examples may wish to skip those sections at the first reading, and proceed directly with Section V. Readers of this document need not know the details of HDM. However, some knowledge of the concepts behind HDM is desirable. There is one example of formal specification in this report, but it is neither a complicated example nor absolutely necessary for the understanding of the system family concept as applied to HDM. For further information, see [10], [11], and [12].

II FAMILIES OF SYSTEMS AND SYSTEM PROPERTIES

A system family is a set of computer systems that possess a common set of system properties. This section contains a definition and general discussion of the term "system property" in relation to the concept of system family and a discussion of various kinds of system properties and the families they generate.

A system property is any statement that can be made about a system. For example, an "n*log(n) sorting algorithm" is a statement of two properties of a program: that it sorts and that it does so using a certain number of operations. The understanding of a system's behavior is usually achieved by examining some of its properties. System properties can be stated in many ways, ranging from natural language to a formal mathematical model. The choice of a medium for stating system properties is of great importance, and is discussed in Section III. In this section, system properties are described in terms of natural language for illustrative purposes.

From the above definition of a system family, it is possible to use a statement of the shared properties as a means for describing the family of systems that share these properties. A similar point of view is proposed by Parnas [13]. The properties that characterize a family are called the generating properties of that family. Thus, a property x generates "the set of all systems with property x." A smaller set of systems is generated by making the properties more restrictive.

A subfamily is any subset of a system family. For example, in the family of Navy message processing systems examined by this project, the NAVMACS series is one subfamily containing several systems, and CUDIXS is another subfamily containing a single system. A subfamily can be generated from a system family by adding properties to the generating

properties of the family. A family of systems can be tree-structured by subfamilies, often a useful exercise in understanding the interrelationship of members of a family.

The kinds of properties that systems in the same family can share fall into three classes:

* Requirements, or external, properties
* Structural properties
* Implementation properties

The first two kinds of properties are reflected in the system design and are called design properties. The last two kinds of properties are independent of external constraints and are called internal properties.

Every system has a set of external constraints, or requirements, that must be reflected in certain properties of the system, called requirements properties. System requirements may be general, such as security or fault-tolerance, or specific, such as running on an AN/UYK-20 or having a tree-structured file system. Requirements affect a system from the outside, and thus must be reflected in the system design at either the top or bottom levels -- those levels that define the external boundaries of the system (the requirements may be reflected in other levels as well). For example, concerning the top level, the requirement to have a LISP system generates the family of all LISP systems. With regard to the bottom level, the requirement to build a system on a 16-bit machine generates the family of all systems that run on a 16-bit machine. Most sets of requirements, however, influence the system design from both top and bottom levels. For example, the set of all message processing systems may require that the user sees a transmission unit called a message, but that the hardware processes a transmission unit known as a packet, and that a message and a packet have a certain relationship (because the data seen by the user must have some relationship to the data actually transmitted by the hardware).

Every systems has some kind of structure, which consists of that system's components and their interrelationship. Structural properties

concern either the components of a system or their interrelationship. Examples of structural properties are having a facility that supports multiple processes (in the case of components) and a hierarchical structure (in the case of a relationship among components). Structural properties need not be visible on the outside of a system (for example, hierarchical structure), although a system's requirements (and thus its external interface) often have a great effect on its structure.

Implementation properties are properties of the code that implements a system. An example of an implementation property for a compiler is that it is implemented in the language it compiles (generally considered an elegant method of building compilers). An example of an implementation property for a table is that it is "hashed," describing a certain way of storing and looking up information (by means of a hash address that is a function of the storage key). The design process, which separates design decisions from implementation decisions, determines whether a property is a design property or an implementation property by including or excluding it from the statement of properties that is known as the system design.

Each of the kinds of properties mentioned above corresponds to given stage of system development. As a system is developed, the set of properties that describes it enlarges. The later properties (such as implementation properties) are often dependent on earlier properties (such as requirements properties), but the converse is not true.

The concept of system property is used currently in the software community for talking about systems, but only in an ad hoc way. Much advantage can be gained by having better methods for stating these properties, as described in Section III.

III STATING SYSTEM PROPERTIES

This section discusses some criteria for choosing a medium in which to express system properties. A medium is a means for conveying or transmitting something. A medium might be a language or a notation, but may have other factors, such as style, involved in its function. The criteria for choosing such a medium are precision, abstraction, restrictiveness, and extensibility. The issue of using a medium that meets these criteria to describe entire families of systems is also discussed.

Any medium for expressing system properties should be precise, that is, formal in the mathematical sense. This criterion reflects the issue of whether or not to use natural language (a medium that is not precise). The advantages of natural language are that it seems easy to understand and requires no special training. However, its disadvantages are that it allows statements of system properties that are ambiguous, incomplete, and inconsistent. Furthermore, the ambiguity, incompleteness, or inconsistency may be difficult, if not impossible, to detect by people or by on-line tools. A precise medium for stating system properties is always unambiguous, and allows the straightforward checking of completeness and consistency. In the long run a precise medium is easier to understand, because it can never be misunderstood.

A medium for stating system properties must allow abstraction, the ability to state as much or as little about the system as is desired, but in a mathematically complete way. For example, for certain purposes it would be desirable to allow the statement of requirements properties only, while omitting all statements concerning structural and implementation properties. The notation of mathematics inherently possesses the essential property of abstraction, allowing a mathematical

model to be built containing exactly what is to be modeled -- no more and no less.

However, a medium for stating system properties should also be somewhat restrictive, based on certain assumptions, such as a model of computation. If the medium for stating system properties allows all possible statements of system properties, it will be very difficult to use, because a user will have to build up an entire set of assumptions before proceeding. The restrictiveness of a more desirable medium results from the fact that certain assumptions (that is, those concerning the model of computation) have already been made, allowing the user to immediately state some meaningful properties, without having to build an entire framework. In addition, restrictiveness of a medium increases the probability that the statement of system properties is correct, because the restrictiveness engenders certain rules against which the statement of properties can be checked. The more checking that can be performed, the greater the chance for correctness.

A medium for stating system properties must be extensible, allowing the user to specify additional properties that are consistent with the properties originally specified. This criterion is essential to maintain consistency if the user is to take a set of generating properties for a family and develop the generating properties of a subfamily or the implementation of a family member.

If the medium for stating system properties is to be used for system families, it must be able to specify the generating properties of a family as well as the additional properties of its subfamilies and of its family members. Given a medium for stating system properties that is to be used for system families and that satisfies the above criteria, there are essentially two approaches to system families. One approach (the one originally proposed for this contract) is to describe the properties of each of the family members explicitly and to show how the sets of properties are related. In this approach, called the enumeration approach, the collection of sets of properties for all

family members is called the superset. The set of properties for a member of the family is called a subset of the superset. The other approach (the one finally decided upon and presented in this document) is to specify only the properties shared by all systems in the family, and to generate the subfamilies and individual family members by adding more properties. In this approach, called the generation approach, the properties shared by all family members are called the family generating properties, the properties shared by a subfamily are called the subfamily properties, and the properties of an individual family member are called family member properties. The enumeration approach is a "bottom-up" method, while the generation approach is a "top-down" method. The generation approach was chosen over the enumeration approach, because the generation approach

* Can specify a family with a large (potentially infinite) number of members via a single design, while the enumeration approach requires as many designs as there are members.
* Is easier to use than the enumeration approach, requiring fewer designs, mechanisms, and tools.
* Would require no cumbersome additions to HDM, while the enumeration approach would.

Thus, the choice of a medium for describing system families is crucial, because the medium can affect the ways in which families are chosen, thus affecting the costs of system development.

IV CHOOSING A SYSTEM FAMILY

This section addresses the important issue of what makes a good system family. If something can be gained (for example, cost savings, understanding) by including several systems in a given family, then it is desirable to do so. Several criteria that indicate a possible gain can be applied:

* The kinds of properties shared
* The ability to generalize the shared properties
* The strength and number of properties shared
* The timing of the decision to specify a system family.

Concerning the kinds of properties that are shared, sets of systems with common design properties make much better system families than do sets of systems with common implementation properties. If design properties are shared, then the resulting systems can share major parts (for example, a user interface or the hardware in the case of requirements properties, or a storage allocation facility in the case of structural properties). Sets of systems that share only implementation properties do not make good families, because the sharing would occur at a smaller scale than an individual component (or else the systems would also share structural properties). This small-scale sharing of code is not addressed directly by the system family methodology presented here. If extensive code sharing is desired, then the structural properties should be chosen to maximize the number of common components, which can each be implemented by the same code. The maximization of structural properties among members of the same family is thus a good practice.

The ability to generalize shared properties is no problem if each member of a family of systems must have <u>all</u> of a set of properties, because the generalization is implicit in the "all." However, if a

member of a family must have at least one of a set of properties, then certain problems can arise in generalizing among these properties. For example, consider the family of systems that can maintain either a priority queue or a binary tree. It is very difficult to find a single property that is a meaningful generalization of the first two. On the other hand, consider the family of systems that can maintain either a priority queue or a queue without priority. These two properties can be generalized to a single one, producing the family of systems that maintain a priority queue, where a queue without priority is simply a priority queue in which all elements have equal priority. Thus, the ability to generalize a set of properties depends on the particular properties involved.

The strength and number of properties that must be shared among the members of a system family determines the closeness of the family. If a small number of weak (or relatively unconstraining) properties are specified, the family will be large and loosely coupled; if a large number of strong (or relatively constraining) properties are specified, the family will be small and tightly coupled. A family that is too loosely coupled (for example, the family of hardware machines that have an "add" instruction) is often not meaningful, because its membership is often too broad to allow members of the family to share entire components. On the other hand, a family that is too tightly coupled (for example, the family of systems that consists of NAVMACS E) may have such a narrow scope as to consist of only one member. In the above two cases the creation of a family is not advisable, because there is little to be gained in so doing. Thus, the criteria of strength and number of properties are often useful in making a good choice of a system family.

It is important to make a decision to incorporate a set of related systems into a family very early in the development stages of the systems involved. If this is not done, the systems in a so-called family may appear to share properties, but because the shared properties were not precisely stated early enough in the design stages, the properties may not correspond exactly. For example, consider two

message processing systems that are independently developed. It may be that the two systems have similar, but slightly different definitions of the terms message, packet, device, and process. The two systems may not actually share these properties, because the systems were developed before the decision was made to combine the systems into a family. It is therefore a difficult exercise to create a family of systems after the fact, and may simply show how the systems in the family should have been designed.

These four criteria for selecting good system family provide some background on the subject of system families, independent of methodology, as an aid in understanding some of the later, more technical discussions.

V SYSTEM PROPERTIES IN HDM

HDM (the SRI Hierarchical Development Methodology) is a convenient vehicle for expressing sets of system properties. The sets of properties can be grouped in many different ways according to three dimensions of the methodology. The first dimension is temporal, the time ordering in which systems are developed according to HDM. The second dimension is structural, concerning the hierarchy of abstract machines that comprise the system design. The third dimension is modular, concerning individual module specifications, irrespective of the structure of any system containing them. This section shows how system properties, described in Section II, can be expressed according to the various dimensions of HDM. Next is a description of how two stages of the methodology, the representation and implementation stages, permit the evolution from a statement of design properties generating a family to a complete system that is a member of that family. In addition, two operations on a design, the addition of levels and the particularization of module specifications, are presented; these operations enable the system designer to proceed from a statement of design properties generating a family to a statement of more restrictive properties generating a subfamily of the original family. Also presented in this section are the following issues: tools to support the family methodology, code sharing in HDM, and limitations of the family methodology.

HDM divides the temporal aspects of system development into three stages: specification, representation, and implementation. Each stage may be viewed as the process of defining certain properties of a system. As one proceeds through these three stages, one defines more and more of the properties of the desired system. Thus, the output of each stage generates a system family that is a subfamily of the family generated by

the output of the previous stage. In the first stage, the specification stage, a specification is written for a hierarchy of abstract machines that comprise the system design. At this point all design properties of the family have been stated. A system specification that generates a family can also be called a family generating design. A specification for a system generates the family of systems that have representations and implementations satisfying it. In the second stage, the representation stage, a formal representation of the data structures of each nonprimitive abstract machine is written in terms of the data structures of the abstract machine at the next lower level. A specification and representation generate the family of systems that have implementations that are consistent with the original specification and representation. In the third stage, the implementation stage, abstract programs are written that implement each nonprimitive abstract machine and that run on the abstract machine at the next lower level. At this point the system is essentially complete, except for particularizations (see below) that may be made and for the compilation of the abstract programs into executable code. Note that the implementation properties described in Section II are defined by both the representation and implementation stages of HDM.

Concerning the structural dimension of HDM, systems that are members of a family may share some parts of a design, but not others. In the structural dimension the generating properties (which can be requirements properties or structural properties) are described by the parts of the design common to all members of the family. A hierarchy in HDM can in most cases be viewed as a sequence of abstract machines, where each abstract machine can be said to occupy a level. For each machine, a formal specification defining its behavior is written. Given a sequence of abstract machine specifications that generates a family, the addition of levels to the sequence generates a subfamily of the original family. For example, the family of systems that have a given user interface may be described by the specification for the abstract machine defining that interface. Conversely, the family of systems that

run on a particular piece of hardware (or operating system) can be viewed as the specifications for a single abstract machine (or sequence of abstract machines) defining that piece of hardware (or operating system). Some families of systems (such as operating systems) may be able to run on different machines and have different user interfaces. Thus they share an internal sub-sequence of abstract machines that generates the family. Examples of relationships between the hierarchies that generate a family and those that generate its subfamilies are shown in Figure 1. In the figure, a family generating design is represented by a sequence of circles, each circle representing an abstract machine, and a subfamily design is produced from the family generating design by adding levels (the sequences of circles in boxes) to the family generating design. The addition of levels is also used in HDM for a single system, when proceeding from a coarse design description to a more detailed design.

Concerning the modular dimension of HDM, each abstract machine or level consists of a set of modules (clusters of related data structures and the operations that access and modify them), each of which is formally specified. A module specification is the smallest unit in HDM in which a set of design properties can be couched. It is possible to specify a module to generate a large family of related system components and to use particularizations of that module specification to generate subfamilies of systems containing that module. Particularizing is the act of taking a module specification and producing a new module specification that describes the behavior of a subclass of the modules originally specified. A particularization is the specification that results from particularizing a module specification. There are four transformations on a module specification that can be invoked to produce a particularization:

* T1 -- instantiating module parameters (symbolic constants that represent either single quantities or functions).
* T2 -- running an auxiliary initialization program that calls the functions of the module.

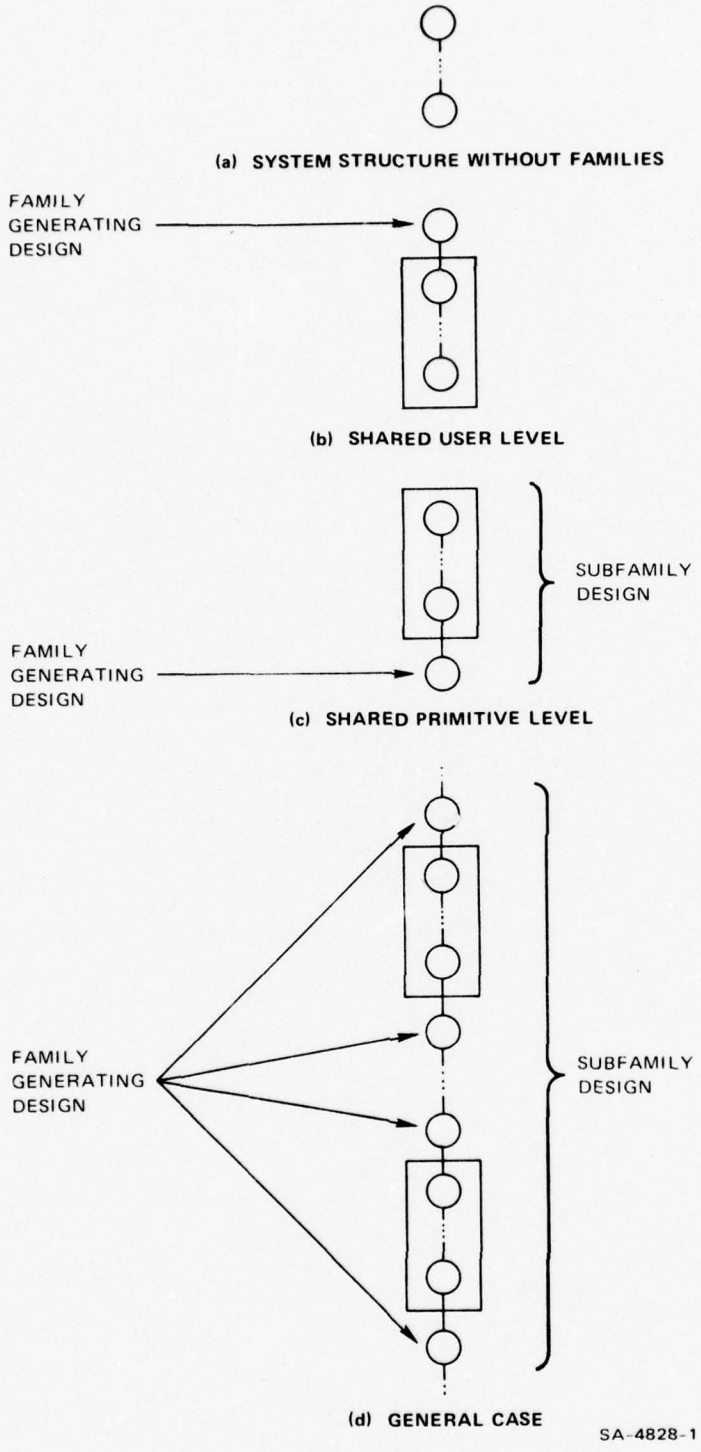

FIGURE 1 SYSTEM FAMILIES AND DESIGN STRUCTURES

* T3 -- substituting constants for actual arguments to visible functions.
* T4 -- forbidding visible functions from being called by user programs.

As an example of how particularization can be applied to module specifications, consider the descriptions of a set of four increasingly generalized modules:

* STACK1 -- a single stack of maximum size 32.
* STACK2 -- a single stack of parametric constant size.
* STACK3 -- a set of stacks (of parametric maximum cardinality), each of parametric constant size, with creation and deletion capability.
* STACK4 -- a set of stacks (of parametric maximum cardinality), whose maximum size can be chosen at creation time from a parametric set of values, with creation and deletion capability.

Their specifications (written in the specification language SPECIAL ([15], [16])) are contained in Tables 1-4 at the end of this section. Note that for all i > 1, the specification of STACKi can be transformed into the specification of STACKi-1, by applying a sequence of the above transformations. STACK2 can be transformed into STACK1 by applying T1 -- instantiating max_stack_size to 32. STACK3 can be transformed into STACK2 by applying T1 (max_number_stacks -> 1), T2 (s0 <- create_stack();), T3 (s -> s0 in all visible functions), and T4 (create_stack, delete_stack). STACK4 can be transformed into STACK3 by applying T1 (max_stack_size_set -> {max_stack_size}). These transformations are a way of generating the design properties of a subfamily by particularization of individual modules in the design. Particularization can also be applied to the representation and implementation stages. Although they are not illustrated here, transformations corresponding to those mentioned above can be performed on the outputs of these later stages.

To summarize, a system specification in HDM is a description of a system family (in terms of design properties) whose members are generated by the stages of representation and abstract implementation.

A description of a subfamily can be generated from a system specification by adding levels to the hierarchy and by particularizing the specifications of its constituent modules.

There are several on-line tools that could be developed to aid in the description and maintenance of system families, in addition to the tools already in existence to support HDM. One possible tool might invoke the transformations for particularizing a given module specification. However, this function can be performed by a conventional text editing program, as it was for the examples given in Tables 1-4. Another possible on-line tool would be a system that keeps track of a family generating design and the design of a few of its subfamilies. Both of these tools might eventually be made part of a complete on-line development environment to support HDM.

Given a family generating design, members of the family may share as much code or as little code as is wished. Of course, particular attention must be given to the design of the family members in order to maximize code sharing. For example, if two designs in a given family are the same except for different particularizations of modules in the family generating design, they may share the same code throughout, because the code can be particularized in the same way as the module specifications can. On the other hand, if two subfamilies of a family have been generated by the addition of levels, identical code may be used only for implementations of levels of the subfamilies that are identical particularizations of the same level in the family generating design, and that have next lower levels that are identical particularizations of the next lower level in the family generating design. Thus, the system family methodology provided by HDM has the potential for saving programming effort as well as design effort, although -- due to performance considerations -- code may not be shared even when possible. In most cases the sharing of design properties is more important than the sharing of code, because it is difficult to share code without sharing components, but it is straightforward to share components without sharing code.

There are some limitations of the system family concept using HDM that prevent one from expressing all useful system families. For example, it is impossible to parameterize a module specification according to type (for example, having a single specification that could describe a stack of objects of any type, depending on the value of a parameter). This is currently impossible because the concept of parameterized types has not been made a part of SPECIAL; however, parameterized types may be added to SPECIAL in the future. In addition, the sharing of small local data structures (such as queues and stacks) has not been directly addressed by the methodology presented here, which deals more with the shared properties of the gross structure of the design as expressed in HDM. However, the module specification technique of HDM allows the accumulation of a central library of the specifications for commonly used small-scale modules. The specifications in the library need be implemented only once, reducing the programming effort.

This section has discussed the ways in which HDM can be used for specifying a system family (the family generating design) and how designs for subfamilies and complete implementations for individual family members can be produced from the family generating design.

Table 1. SPECIFICATION OF STACK1

```
MODULE stack1

    FUNCTIONS

VFUN ptr() -> INTEGER i;
    HIDDEN;
    INITIALLY i = 0;

VFUN stackval( INTEGER i) -> INTEGER v;
    HIDDEN;
    INITIALLY v = ?;

OFUN push( INTEGER v);
    EXCEPTIONS ptr() >= 32;
    EFFECTS
      'ptr() = ptr() + 1;
      'stackval( 'ptr()) = v;

OVFUN pop() -> INTEGER v;
    EXCEPTIONS ptr() = 0;
    EFFECTS
      'stackval( ptr()) = ?;
      'ptr() = ptr() - 1;
      v = stackval( ptr());

END_MODULE
```

Table 2. SPECIFICATION OF STACK2

```
MODULE stack2

    PARAMETERS

INTEGER max_stack_size;

    ASSERTIONS

max_stack_size > 0;

    FUNCTIONS

VFUN ptr() -> INTEGER i;
    HIDDEN;
    INITIALLY i = 0;

VFUN stackval( INTEGER i) -> INTEGER v;
    HIDDEN;
    INITIALLY v = ?;

OFUN push( INTEGER v);
    EXCEPTIONS ptr() >= max_stack_size;
    EFFECTS
      'ptr() = ptr() + 1;
      'stackval( 'ptr()) = v;

OVFUN pop() -> INTEGER v;
    EXCEPTIONS ptr() = 0;
    EFFECTS
      'stackval( ptr()) = ?;
      'ptr() = ptr() - 1;
      v = stackval( ptr());

END_MODULE
```

Table 3. SPECIFICATION OF STACK3

```
MODULE stack3

    TYPES

stack_type: DESIGNATOR;

    PARAMETERS

INTEGER max_number_stacks;
INTEGER max_stack_size;

    ASSERTIONS

max_number_stacks > 0;
max_stack_size > 0;

    FUNCTIONS

VFUN stack_exists( stack_type s) -> BOOLEAN b;
    HIDDEN;
    INITIALLY b = FALSE;

VFUN ptr(stack_type s) -> INTEGER i;
    HIDDEN;
    INITIALLY i = ?;

VFUN stackval( stack_type s; INTEGER i) -> INTEGER v;
    HIDDEN;
    INITIALLY v = ?;

OVFUN create_stack() -> stack_type s;
    EXCEPTIONS
      CARDINALITY({ stack_type s1 ¦ stack_exists(s1)}) >=
        max_number_stacks;
    EFFECTS
      s = NEW( stack_type);
      'stack_exists(s);
      'ptr(s) = 0;

OFUN delete_stack( stack_type s);
    EXCEPTIONS NOT stack_exists(s);
    EFFECTS
      NOT 'stack_exists(s);
      'ptr(s) = 0;
      FORALL INTEGER i: 'stackval(s, i) = ?;
```

Table 3 (continued)

```
OFUN push( stack_type s; INTEGER v);
    EXCEPTIONS
      NOT stack_exists(s);
      ptr(s) >= max_stack_size;
    EFFECTS
      'ptr(s) = ptr(s) + 1;
      'stackval(s,  'ptr(s)) = v;

OVFUN pop( stack_type s) -> INTEGER v;
    EXCEPTIONS
      NOT stack_exists(s);
      ptr(s) = 0;
    EFFECTS
      'stackval(s,  ptr(s)) = ?;
      'ptr(s) = ptr(s) - 1;
      v = stackval(s, ptr(s));

END_MODULE
```

Table 4. SPECIFICATION OF STACK4

```
MODULE stack4

    TYPES

stack_type: DESIGNATOR;

    PARAMETERS

INTEGER max_number_stacks;
SET_OF INTEGER max_stack_size_set;

    ASSERTIONS

max_number_stacks > 0;
CARDINALITY( max_stack_size_set) > 0;
FORALL INTEGER size INSET max_stack_size_set: size > 0;

    FUNCTIONS

VFUN stack_exists( stack_type s) -> BOOLEAN b;
    HIDDEN;
    INITIALLY b = FALSE;

VFUN ptr(stack_type s) -> INTEGER i;
    HIDDEN;
    INITIALLY i = ?;

VFUN max_stack_size( stack_type s) -> INTEGER size;
    HIDDEN;
    INITIALLY size = ?;

VFUN stackval( stack_type s; INTEGER i) -> INTEGER v;
    HIDDEN;
    INITIALLY v = ?;

OVFUN create_stack( INTEGER size) -> stack_type s;
    EXCEPTIONS
      NOT size INSET max_stack_size_set;
      CARDINALITY({ stack_type s1 ¦ stack_exists(s1)}) >=
        max_number_stacks;
    EFFECTS
      s = NEW( stack_type);
      'stack_exists(s);
      'ptr(s) = 0;
      'max_stack_size(s) = size;
```

Table 4 (continued)

```
OFUN delete_stack( stack_type s);
    EXCEPTIONS NOT stack_exists(s);
    EFFECTS
      NOT 'stack_exists(s);
      'ptr(s) = 0;
      'max_stack_size(s) = ?;
      FORALL INTEGER i: 'stackval(s, i) = ?;

OFUN push( stack_type s; INTEGER v);
    EXCEPTIONS
      NOT stack_exists(s);
      ptr(s) >= max_stack_size(s);
    EFFECTS
      'ptr(s) = ptr(s) + 1;
      'stackval(s,  'ptr(s)) = v;

OVFUN pop( stack_type s) -> INTEGER v;
    EXCEPTIONS
      NOT stack_exists(s);
      ptr(s) = 0;
    EFFECTS
      'stackval(s,  ptr(s)) = ?;
      'ptr(s) = ptr(s) - 1;
      v = stackval(s, ptr(s));

END_MODULE
```

VI APPLICATION OF THE SYSTEM FAMILY CONCEPT

This section briefly describes the experience of the personnel of SRI International in applying the methodology for system families to an example application, a family of message processing systems (MPSs). The purpose of this example is to test the system family methodology on a problem of relevance to the Navy, hopefully demonstrating relevance and general effectiveness as well. The problem entails describing the family of systems consisting of two different kinds of MPSs, the NAVMACS family and the CUDIXS system. See [17] for a description of the facilities provided by these two systems. The work has involved the preliminary design and choice of functions for the family generating design [18] and will ultimately include the full specifications for the family generating design, now being produced. Because the design effort is still in progress, the conclusions to be drawn from the example must of course be tentative.

The problem was approached as follows: first, all the necessary requirements properties were stated; next, the structural properties were determined; and finally, the family generating design was constructed on the basis of the list of requirements properties and structural properties. The rest of this section contains observations and conclusions that have resulted from this effort to date. The first class of observations concerns the properties that were chosen for the family generating design and how they were chosen. The second class of observations concerns the way in which the design changed as it became more precise, and why it did so.

Concerning requirements properties, it is important to be able to determine which requirements to include and which to leave out. Those requirements included in the family generating design were

* The existence of a network

* Multiple addresses at a node of a network
* Messages
* Communications hardware
* Packets
* Choice of CPU (AN/UYK-20)
* The existence of other applications on the system
* Journals, logs, and reports.

Those items left out of the requirements were

* Formats for messages and packets
* Algorithms and strategies for processing data
* Internal data -- its content and structure.

It is possible to generalize from these specific decisions. A necessary requirement may be any of the following:

* All information transmitted to and received from users
* All hardware information
* How the system fits into its environment (in this case a network).

An unnecessary requirement may be any of the following:

* Data format
* An algorithm for processing data
* Any internal information.

Much detail may be abstracted away in the definition of necessary requirements. For example, the destinations of a message are determined by examining an explicit address, together with some keywords and a Standard Subject Information Code (SSIC). In the family generating design, all of this data may be combined into the single field "address specification," without being more precise. At a later stage in the design, this data may be further specified.

Concerning structural properties, the following have been included in the family generating design:

* The support of multiple processes
* Storage allocation
* Packet and message validation

* Virtual I/O
* Message routing.

As in the case of requirements properties, items of data formats and algorithms have not been included in the family generating design. In general, properties pertaining to mechanism (for example, the system facility for labeling a subset of all messages as "valid") have been included, while those pertaining to policy (for example, the manner whereby the system decides whether or not a given message is valid) have been omitted. As another example of mechanism versus policy, the specification of a storage allocation module describes the ability to secure and to free storage, but does not specify how this is to be done. Some small-scale data structures, although they might be shared among members of the family, were left out because they were not necessary; they could be built directly out of the mechanisms provided. Examples of such data structures are stacks, queues, and tables. However, specifications for such commonly used modules could be provided in a general library (as described in Section V) for designers to use at a later stage in the design, because designers should not have to repeat the writing of specifications and implementations for commonly used modules. This issue is separate from the issue of system families.

As the family generating design was formalized, several aspects of the design changed. Among the changes were

* The use of a simple scheme for multiple processes rather than a complicated multiple process scheme or a single process.
* The use of a simple storage allocation algorithm.
* The simplification of I/O.

Other changes in the design are also tentatively planned, and more changes may arise as the specifications are completed. As a result of the changes made to date, the design has become simpler than it was originally. Change is common in the use of HDM, because one's view of a problem changes as the problem (and its solution) is mathematically formalized. However, the effects of such changes are usually limited because of the structuring of the system into levels and modules.

The results of this effort so far have been the production of a good design and the ability to understand the design implications of the spectrum of properties related to the system family. From this effort, it can be tentatively concluded that the design of the defining system of the MPS family is much simpler than originally anticipated. This simplification may well reduce costs if the system family were to be implemented. It can also be concluded that designing a set of systems to be a family is preferable to combining a set of existing systems, after the fact, into a family, because the shared properties of existing systems, although similar, often do not match exactly.

VII RELEVANCE OF THE SYSTEM FAMILY CONCEPT TO THE SOFTWARE PROBLEM

Even if the problem of system families can be handled by HDM, is this problem -- in the Navy, for example -- so pervasive that it should require a particular set of techniques to address it? This section supports the conjecture that the class of problems for which families of systems are a solution is the same as the class of problems for which a single system is the solution. Thus, the same approach should be used in either case, and the decision on whether the ultimate product will be a single system or a family of systems should be postponed until the family generating design is completed. The family methodology is generally useful because it attacks the problem of understanding the requirements and their relationship to the design properties of the system (or systems) being developed.

Using conventional methods, large software projects are conceived and developed according to the following scenario. The system planners start with a problem or a set of related problems for which a range of system requirements is determined (for example, the set of message processing requirements for the Navy); these requirements are usually stated in a document by means of some medium that is not mathematically precise. The planners would prefer, for reasons of cost, that a single system could be built that solves the entire range of problems. In many cases this cannot happen (because the range of requirements are too broad to be addressed by one system), so the system planners set out to partition the range of requirements at a very early stage in the software development process, and to develop a separate system to address the requirements of each element of the partition. This partitioning is usually done too early in the software development process, resulting in a set of loosely connected systems that often do not, in toto, successfully address the entire range of requirements.

Either there is too much overlap, or some areas of the range of requirements are not addressed at all, and the result is increased cost.

Instead of the conventional approach to system planning and development, the family approach is proposed. The initial effort would be to construct a single design, the family generating design, that can be used to address the full range of requirements. The choice of the individual family members can be postponed until after the family generating design is completed, at which time enough is known about the particular requirements to make a more intelligent decision concerning the choice of family members. As an added benefit, much of the design -- for all family members -- will have been completed at the time that the family members are chosen. Even if only one system is ultimately developed, no effort will have been wasted. In fact, the simplification that results in the ultimate construction of a single system has probably reduced the costs by building one system instead of several. If the requirements change in midstream, a new family member that meets these changed requirements can often be developed at reduced cost. Furthermore, if an entire system family is to be developed, the family members can share large parts of the implementation code, reducing costs even further.

VIII CONCLUSIONS

This document has presented an approach to the design and implementation of families of systems that

* Is based on a design methodology (HDM) that is useful in its own right for designing single systems.
* Requires minimal additions to existing technology in order to be useful.
* Is being tested in an applications area of relevance to the Navy.

The approach is as follows: first, produce a single hierarchical design specification (according to HDM), called the family generating design, which contains the properties shared by all members of the family; then, the design of a subfamily or an implementation of an individual family member can be generated by adding properties that are particular to that subfamily or family member.

The method has great potential for reducing costs in the design through implementation stages of a predetermined set of related systems. Furthermore, the family approach is believed to be useful in reducing costs when applied at the earliest stages of system development (for example, the planning stages), when only the requirements are known, even before the exact number and range of systems in the family are chosen. At this early a stage in system development, the family methodology could have an effect on the number of systems ultimately developed, which has the greatest potential for cost savings.

REFERENCES

1. D. L. Parnas, "Information Distribution Aspects of Design Methodology," Information Processing 71, pp. 339-344 (North-Holland Pub. Co., Amsterdam, 1972).

2. D. L. Parnas, "A Technique for Software Module Specification with Examples," Comm. ACM, Vol. 15, No. 5, pp. 330-336 (May 1972).

3. D. L. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," Comm. ACM, Vol. 15, No. 12, pp. 1053-1058 (December 1972).

4. E. W. Dijkstra, "Complexity Controlled by Hierarchical Ordering of Function and Variability," in Report on a Conference on Software Engineering, Randell and Naur, eds. (NATO, 1968).

5. E. W. Dijkstra, "Notes on Structured Programming," in Structured Programming, C. A. R. Hoare, ed. (Academic Press, New York, 1972).

6. C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," Comm. ACM, Vol. 12, No. 10 (October 1969).

7. C. A. R. Hoare, "Proof of correctness of Data Representations," Acta Informatica, Vol. 1, pp. 271-281 (1972).

8. L. Robinson, K. N. Levitt, P. G. Neumann, and A. R. Saxena, "On Attaining Reliable Software for a Secure Operating system," Proc. Int. Conf. on Reliable Software, 13-15 April 1975, in SIGPLAN Notices, Vol. 10, No. 6, pp. 267-284 (June 1975). Also appears as, "A Formal Methodology for the Design of Operating System Software," in Current Trends in Programming Methodology, Vol. I, R. T. Yeh, ed. (Prentice-Hall, New York, May 1977).

9. L. Robinson and K. N. Levitt, "Proof Techniques for Hierarchically Structured Programs," Comm. ACM, Vol. 20, No. 4, pp. 271-283 (April 1977).

10. K. N. Levitt, W. Kautz, and L. Robinson, "HDM Handbook," to be published as a Technical Report, Computer Science Laboratory, SRI International, Menlo Park, CA (1977).

11. L. Robinson, "HDM -- Concepts and Facilities," to be published as a Technical Report, Computer Science Laboratory, SRI International, Menlo Park, CA (1977).

12. L. Robinson, "HDM -- Command and Staff Manual," to be published as a Technical Report, Computer Science Laboratory, SRI International, Menlo Park, CA (1977).

13. D. L. Parnas, "On the Design and Development of Program Families," IEEE Trans. Software Engineering, Vol. 2, No. 1, pp. 1-9 (March 1976).

14. A. N. Habermann, L. Flon, and L. Cooprider, "Modularization and Hierarchy in a Family of Operating Systems," Comm. ACM, Vol. 19, No. 5, pp. 266-272 (May 1976).

15. L. Robinson and O. Roubine, "SPECIAL -- a SPECIfication and Assertion Language," Technical Report CSL-46, Computer Science Laboratory, SRI International, Menlo Park, CA (January 1977).

16. O. Roubine and L. Robinson, "SPECIAL Reference Manual," Technical Report CSL-45, Computer Science Laboratory, SRI International, Menlo Park, CA (January 1977).

17. J. H. Wensley, R. E. Keirstead, and H. M. Zeidler, "Message Processing System Facilities with Respect to NAVMACS and CUDIXS," Technical Report, SRI International Project 4828, SRI International, Menlo Park, CA (November 1976).

18. L. Robinson and L. Chaitin, "Design of a Family of Message Processing Systems," to be published as a Technical Report, Computer Science Laboratory, SRI International, Menlo Park, CA (1977).

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)

The Relationship of System Families to HDM

5. TYPE OF REPORT & PERIOD COVERED

Technical Report

6. PERFORMING ORG. REPORT NUMBER

Technical Report CSL-50

7. AUTHOR(s)

Lawrence Robinson

8. CONTRACT OR GRANT NUMBER(s)

N00123-76-C-0195

9. PERFORMING ORGANIZATION NAME AND ADDRESS

SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

Deliverable A-013

11. CONTROLLING OFFICE NAME AND ADDRESS

Naval Ocean Systems Center
San Diego, CA 92152

12. REPORT DATE

July 1977

13. NO. OF PAGES

34

14. MONITORING AGENCY NAME & ADDRESS (if diff. from Controlling Office)

15. SECURITY CLASS. (of this report)

Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

System Family, Formal Specification, Methodology

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A system family is a set of computer systems that share common properties. This report discusses the use of an existing software development methodology, HDM (the SRI Hierarchical Development Methodology), to describe families. It is believed that, if a group of related systems is developed as a family using this methodology, costs will be saved over separate development efforts. Several issues in system families are presented, and some ongoing work in applying this methodology is discussed.

DISTRIBUTION LIST

Defense Documentation Center Cameron Station Alexandria, VA 22314	12 copies
Mr. Tony Allos Code 6201 Naval Ocean Systems Center 271 Catalina Boulevard San Diego, CA 92152	1 copy
Mr. L. Sutton Code 8223 Naval Ocean Systems Center 271 Catalina Boulevard San Diego, CA 92152	35 copies
Mr. William Carlson Advanced Research Projects Agency Office of Secretary of Defense 1400 Wilson Boulevard Arlington, VA 22209	15 copies
Mr. Neal Hampton Code 8223 Naval Ocean Systems Center 271 Catalina Boulevard San Diego, CA 92152	15 copies
Professor Stuart Madnick MIT - Sloan School E53-330 Cambridge, MASS 02139	1 copy
Mr. John Machado The Naval Electronic Systems Command National Center No. 1 Crystal City, Washington, D.C. 20360	5 copies